Load Balancing of Java Applications by Forecasting Garbage Collections

A. Omar Portillo-Dominguez*, Miao Wang*, Damien Magoni†, Philip Perry*, and John Murphy*
*Lero, School of Computer Science and Informatics, University College Dublin, Ireland
†LaBRI – CNRS, University of Bordeaux, France
e-mail: andres.portillo-dominguez@ucdconnect.ie, miao.wang@ucd.ie,
magoni@labri.fr, philip.perry@ucd.ie, j.murphy@ucd.ie

***Abstract*—Modern computer applications, especially at enterprise-level, are commonly deployed with a big number of clustered instances to achieve a higher system performance, in which case single machine based solutions are less cost-effective. However, how to effectively manage these clustered applications has become a new challenge. A common approach is to deploy a front-end load balancer to optimise the workload distribution between each clustered application. Since then, many research efforts have been carried out to study effective load balancing algorithms which can control the workload based on various resource usages such as CPU and memory. The aim of this paper is to propose a new load balancing approach to improve the overall distributed system performance by avoiding potential performance impacts caused by Major Java Garbage Collection. The experimental results have shown that the proposed load balancing algorithm can achieve a significant higher throughput and lower response time compared to the round-robin approach. In addition, the proposed solution only has a small overhead introduced to the distributed system, where unused resources are available to enable other load balancing algorithms together to achieve a better system performance.**

I. Introduction and Related Work

Enterprise applications commonly require to achieve fast response time and high throughput to constantly meet their service level agreements. These applications make wide use of variants of distributed architectures, usually using some form of load balancing to optimise their performance. Since then researchers have made efforts to improve the business intelligence of load balancers to effectively manage workloads. For example, the authors of [11] proposed a technique to estimate the global workload of a load balancer to use this information in the balancing of new workload. Meanwhile, the work on [5] presented a framework for processor load balancing during the execution of application programs. Regarding Java technologies, the authors of [2] enhanced a load balancing algorithm for Java applications by considering the utilisation of the JVM threads, heap and CPU to decide how to distribute the load. Similarly the work in [6] proposes a function to calculate the utilisation of an Enterprise JavaBean (EJB) and then uses this information to balance the load among the available EJB instances. However, Garbage Collection (GC) metrics have not been considered so far. This gap offers an interesting niche which is yet to be exploited.

GC is a core feature of Java which automates most of the tasks related to memory management. However, when the GC is triggered, it has an impact on the system performance by pausing the involved programs. Even though milliseconds pauses caused by GC does not necessarily lead to a harmful problem, delays of hundreds of milliseconds, let alone full seconds, can cause trouble for applications requiring fast response time or high throughput. This is more likely to occur in the Major Garbage Collection (MaGC), which has the most expensive type of GC pauses [15].

Many research studies have provided evidence to quantify the performance costs of the GC. For example, in [18] authors identified the GC as a major factor degrading the behaviour of a Java Application Server (a traditional Java business niche) due to the sensitivity of the GC to the workload. In these experiments the GC took up to 50% of the execution time of the Java Virtual Machine (JVM), involving pauses as high as 300 seconds. The MaGC represented 95% of those pauses on the heaviest workload. Similarly, a survey conducted among Java practitioners [14] reported GC as a typical area of performance issues in the industry. For these reasons, it is commonly agreed that the GC plays a key role in the performance of Java systems.

The goal of this work is to predict the MaGC events and use this information in the decision making process of a load balancer to improve the system performance. Our solution consists of two algorithms. A load balance algorithm which avoids sending any incoming workloads to the application nodes which are likely to suffer MaGC, and a forecast algorithm to predict the MaGCs. The experiment results have shown that this strategy offers a significant performance gain: The average response time of the tested applications decreased between 74% and 99%, while the average throughput increased between 4% and 51%.

In summary, the contributions of this paper are:

1) A novel load balance algorithm that uses MaGC forecasts to improve the performance of distributed Java systems.
2) A novel forecast algorithm that enables Java systems to predict when a MaGC event will occur.
3) A validation of the algorithms consisting of a prototype and two experiments. The first proves the accuracy of the MaGC forecast. The second demonstrates the performance benefits of using the forecast for load balancing.

II. Background

Memory Management in Java. GC is a form of automatic memory management which offers significant software engineering benefits over explicit memory management: It frees programmers from the burden of manual memory management, preventing the most common sources of memory leaks and overwrites [17], as well as improving the programmer's productivity [9]. Despite these advantages, the GC comes with a cost (as discussed in Section I).

Nowadays the most common heap type in Java is the generational heap[1], where the objects are segregated by age into memory regions called generations. New objects are created in the Youngest generation. The survival rates of younger generations are usually lower than those of older ones, meaning that younger generations are more likely to be garbage and can be collected more frequently than older ones. The GC in the younger generations is known as Minor GC (MiGC). It is usually inexpensive and rarely causes a performance concern. MiGC is also responsible of moving the live objects which have become old enough to the older generations, meaning that the MiGC plays a key role in the memory allocation of older generations. The GC in the older generations is known as MaGC and is commonly accepted as the most expensive GC due to its performance impact[15].

Also, it is not possible to programmatically force the execution of the GC [7]. The closest action a developer can perform is to call the method *System.gc()* to suggest the JVM to execute a MaGC. However, the JVM is not forced to fulfill this request and may choose to ignore it. The usage of this method is discouraged by the JVM vendors[2] because the JVM usually does a much better job on deciding when to do GC.

Garbage Collection Optimisation & Memory Forecast. Multiple research works have proposed new GC algorithms [3], [4], [10], [12] that have smaller performance impacts on the applications. Even though all these works have helped to reduce the impact of the MaGC, GC remains a concern due to the different factors that can affect its performance.

Memory forecast is also an active research topic, looking for ways to invoke a GC only when it is worthwhile. For example, the work presented in [16] exploits the observation that dead objects tend to cluster together to estimate how much space would be reclaimable to avoid low-yield GCs. However memory forecasts alone do not provide enough information to know when the next MaGC would occur.

III. Proposed Solution

A. Use case: Adaptive Load Balancer

In a distributed Java system, it is preferable that the occurrence of MaGCs in the individual nodes do not affect the performance of the system. To achieve this goal, a system can take different actions. For instance, a system might change its workload schedule to avoid the impact of the MaGCs or encourage a MaGC when a resource load (i.e. CPU) is low.

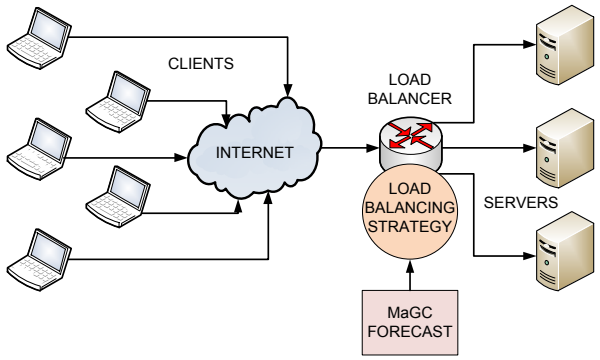

Fig. 1. Adaptive Load Balancer

Among the potential use cases, our work centered on enhancing the performance of a load balancer. This use case was selected because variants of this distributed architecture are commonly used at enterprise-level. This scenario is shown in Figure 1, where the load balancer selects those nodes which are less likely to suffer a MaGC pause as optimal nodes for given workloads. This strategy can keep the system performance safe from any major MaGC pauses.

B. Major GC Forecast Algorithm

The next sections describe our proposed forecast algorithm. The below definitions will be used on the algorithm discussion:

Time is always expressed as the number of milliseconds that have passed since the application started.

Young/Old Generation Samples are composed of a timestamp and the usage of the corresponding memory generation.

MiGC sample is composed of the start time, the end time and the memory usage before and after the latest MiGC event.

Observations are used in a statistical context and are composed of one independent and one dependent values. When the dependent value does not contain historical data, the observation is referred as a forecast observation.

Steady state is the state an application reaches after the JVM finishes loading all its classes. It is assumed that this state has been reached if the number of loaded classes remain unchanged for a certain number of consecutive samples.

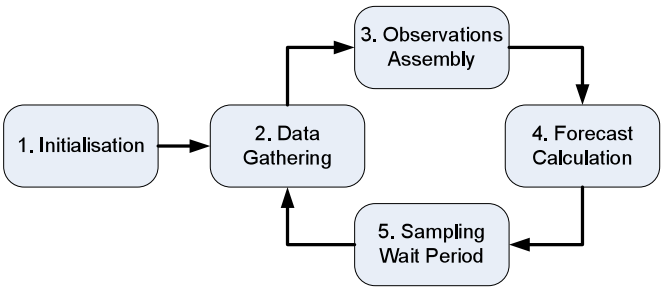

Fig. 2. MaGC Forecast Process - Overview

1) Algorithm Overview: Figure 2 depicts an overview of the algorithm, which is composed of five main phases. First the *Initialisation* which sets the parameters required by the algorithm. After it occurs, the other phases are iteratively

[1]http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf

[2]http://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/devapps/-codeprac.html

done to produce MaGC forecasts continuously: New samples are retrieved from the monitored JVM in the *Data Gathering* phase. Then new observations are generated using the new samples in the *Observations Assembly* phase. Next the *Forecast Calculation* occurs. Finally, the logic awaits a *Sampling Interval* before a new iteration starts. This loop continues until the monitored application finishes.

Our algorithm is designed to work on generational heaps, as it is the most common type of Java heap. It only uses standard data that can be obtained from any JVM (such as GC) to make it easy to implement either within or outside the JVM. If the algorithm is implemented within the JVM, the interaction with potential consumers would be simplified. If it is implemented outside the JVM, the implementation would work with any JVM currently available, facilitating the adoption.

2) Detailed Algorithm: It is presented in Algorithm 1, and its phases are explained in the following sections.

Algorithm 1: MaGC Forecast

```
Input: Sampling Interval, Forecast Window Size,
       Warm-up Window Size
Output: Forecast time of the next MaGC event
1  steadyState := not reached
2  while forecast is needed do
3      Get new OldGen sample
4      if steadyState is not reached then
5          Get new loaded classes sample
6          if warm-up period is over then
7              steadyState := reached
8      Get new MiGC sample
9      Calculate new memory deltas
10     Update memory totals
11     Generate new observations
12     if steadyState is reached then
13         Forecast memory pending to be allocated
14         Forecast time of the next MaGC event
15     Wait the Sampling Interval
```

Initialisation. Here the configuration parameters are set:

- *Sampling Interval*: How often the samples are collected.
- *Forecast Windows Size (FWS)*: How many observations are used as historical data in the forecast calculation.
- *Warm-up Window Size*: How many samples are used to determine if the application has reached its *steady state*.

Data Gathering. Its objective is to capture an updated snapshot of the monitored JVM. It starts by collecting a new *Old Generation* sample. Then, if the application has not reached the *steady state* yet, a new *loaded classes* sample is collected and its history is reviewed. If the warm-up period is over, a flag is set to indicate this. Later a new MiGC sample is collected and added to the MiGC history. After having samples from at least two MiGCs, the next metrics are calculated:

- *Time between MiGCs* ($\Delta\ T_{MiGC}$): How much time elapsed between the latest two MiGCs.
- *YoungGen Memory Allocation* ($\Delta\ YMA_{MiGC}$): How much memory was used to create new objects between the latest two MiGCs.
- *OldGen Memory Allocation* ($\Delta\ OMA_{MiGC}$): How much OldGen Allocation occurred because of the latest MiGC (meaning that some objects have became old enough to be moved to the OldGen by the latest MiGC).

The above metrics are added up into their respective totals (e.g., *Total Time between MiGCs*) to keep track of how the metrics grow through time. This data is the key input of the regression models used by the algorithm, as explained below.

Observations Assembly. Two types of observations are generated and added to their histories. Each is composed of one independent (y axis) and one dependent (x axis) values: The first type (YoungGen-OldGen) captures the relationship between the memory allocation rates (MAR) in the Young and Old Generations. This captures how the Old Generation grows (eventually leading to a MaGC) in relation to the object allocations requested by the application (which occur in the Young Generation). In this observation the dependent value is the *Total YoungGen Memory Allocation* and the independent value is the *Total OldGen Memory Allocation*. The second type of observation (Time-YoungGen) captures the relationship between the time and the Young MAR. Here the dependent value is the *Total Time between MiGCs* and the independent value is the *Total YoungGen Memory Allocation*.

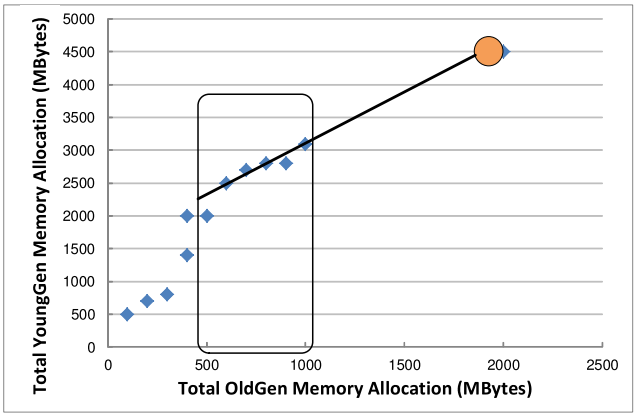

Fig. 3. Old memory exhaustion forecast

Forecast Calculation. This phase first evaluates if the application has reached the *steady state*. If so, two projections are calculated using linear regression models (LRM). The first projection corresponds to how much memory allocation needs to occur in the Young Generation before the free memory in the Old Generation gets exhausted (hence triggering a MaGC). This is calculated by initializing a LRM with a subset of *YoungGen-OldGen* observations (defined by the FWS) and then feeding the LRM with a forecast observation whose independent value is the sum of the current *Total OldGen Allocation* and the free *OldGen* memory. This is shown in Figure 3. In this example, the free *OldGen* memory is 1,000MB. As our *Total OldGen Allocation* is also 1,000MB, the independent value of our forecast observation is 2,000MB. Using the observations within the FWS (the rounded rectangle), the LRM

predicts how much memory allocation needs to occur in the YoungGen before the next MaGC occurs (4,500MB).

The second projection is the core output of this algorithm: The MaGC forecast time. It is calculated by initializing a LRM with a subset of *Time-YoungGen* observations and feeding it with a forecast observation whose independent value is the result of the first projection. This is represented in Figure 4. Using the observations within our FWS, the LRM predicts when the necessary memory allocation in the YoungGen will occur (4,500MB in our example), consequently triggering the next MaGC (around the millisecond 13,000 in our example).

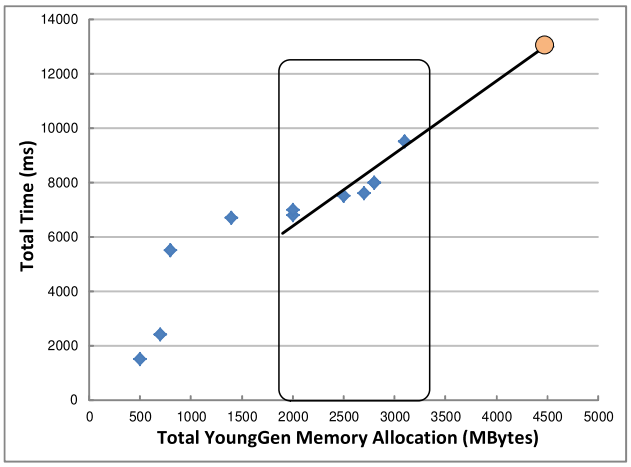

Fig. 4. MaGC event forecast

Sampling Wait Period. Finally, the process waits the number of milliseconds configured in the *Sampling Interval* before starting the next round of iterative steps of the algorithm.

C. MaGC-Aware Load Balancing

To assess the performance benefits that can be achieved by adapting the load balancing based on the MaGC forecast information, we modified the well-known round robin load balancing algorithm[3]. Our proposed algorithm is presented in Algorithm 2. It requires two inputs: The *Number of available nodes* from which the algorithm will select the next node to send workload; and the *MaGC Threshold*, which is the time threshold when a node stops being considered a feasible candidate because the next MaGC is too close. For example, if the *MaGC Threshold* is 5 seconds and the current time is 4:00:00PM, any nodes which report a MaGC forecast between 4:00:00PM and 4:00:05PM will be skipped as their forecasts fall within the configured MaGC Threshold.

When compared against the normal round robin, our algorithm has two differences. The main one is that it performs an additional check to adapt the selection of the next node to a close MaGC event. This check reviews if the pre-selected node (as per the normal round-robin logic) will suffer a MaGC within the *MaGC Threshold*. If it does, the node is skipped and the next available node is evaluated (lines 11 to 15). The second change is an escape condition (the *forecastTries* variable) which counts the number of evaluated nodes to prevent an infinite loop in case all nodes are about to suffer a MaGC within the *MaGC Threshold*. If this occurs, the algorithm would behave as a normal round robin algorithm.

Algorithm 2: MaGC-Aware Load Balancing

```
   Input: Number of available nodes avNodes, MaGC
          Threshold maGCThres
   Output: Next available node (nextNode)
1  indexNextNode := 0
2  forecastTries := 0
3  while load balance adaptiveness is needed do
4      nextNode := undefined
5      while nextNode is undefined do
6          indexNextNode := indexNextNode+1
7          if indexNextNode >avNodes then
8              indexNextNode := 1
9          nextNode := indexNextNode
10         if forecastTries <avNodes then
11             Get MaGC forecast of server
               indexNextNode
12             remainingTime := forecast Time - current
               time
13             if remainingTime <=maGCThres then
14                 nextNode := undefined
15                 forecastTries := forecastTries+1
16             else
17                 forecastTries := 0
18         else
19             forecastTries := 0
20     use nextNode for the next workload
```

D. Prototype Implementations

MaGC Forecast Algorithm. This prototype was developed external to the JVM, using Java Management Extension (JMX)[4] to interact with the monitored JVM. This technology was chosen because it is a standard component of Java which can retrieve all needed information (e.g., GCs).

MaGC-Aware Load Balancing Algorithm. This prototype was built on top of the Central Directory[5], which is a light-weight load balancer. This solution was chosen because it is open source and developed in Java, characteristics which facilitated its integration with the MaGC forecast prototype.

IV. Experimental Evaluation

A. Experiment #1: MaGC Forecast Accuracy

Environment. All experiments were performed in a virtual machine (VM) equipped with 3 CPUs, 10GB of RAM, and 50GB of HD; Linux Ubuntu 12.04L 64-bit, and Oracle Hotspot JVM 7. The JVM was configured to initialise its *Java Heap* to its maximum size to keep it constant during the experiments. The calls to programmatically request a MaGC were disabled.

[3]http://publib.boulder.ibm.com/infocenter/wsdatap/4mt/topic/com.ibm.dp.xa.doc/administratorsguide.xa35263.htm

[4]http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html

[5]http://javalb.sourceforge.net/

Java Benchmarks. The DaCapo[6] benchmark 9.12 was chosen because it stresses the GC system more than other benchmarks (as proved in [1]) and it also offers a wide range of application behaviours to test. For each benchmark, the largest *Sample size* was used (among the available pre-defined sizes[7]). Also different *Number of iterations* (in increments of 5) and *Heap sizes* (in increments of 50MB) were tried until achieving successful executions that triggered MaGCs. These configurations are summarized in Table I.

TABLE I
DACAPO CONFIGURATIONS

Benchmark	Sample Size	#Iters	Heap Size(MB)
avrora	large	30	100
batik	large	60	50
eclipse	large	5	800
h2	huge	5	1600
pmd	large	50	400
sunflow	large	80	200
tomcat	huge	10	100
tradebeans	huge	5	800
tradesoap	huge	5	800
xalan	large	40	50

Also, a *Warm-up timeframe* of 5 seconds was found to be big enough to allow all programs to finish loading their classes before the first forecast was generated.

MaGC Forecast Algorithm parameters. As explained in Section III-B, this algorithm requires 3 parameters. To evaluate the behaviour of the algorithm to the FWS, a broad range of values was tested (2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048 and 4096). A value of 100ms was selected as *Sampling Interval*, assuming that no more than one GC would occur within that timeframe (hence not missing to sample any GC). Finally, a *Warm-up Window Size* of 50 was used (the result of dividing the *Warm-up timeframe* by the *Sampling Interval*).

GC strategies. Three of the most commonly used GC strategies[8] in the industry were selected: Serial GC is preferable for client JVMs, Parallel GC is better for server JVMs except when response time is more important than throughput. If so, Concurrent GC is preferred.

Metrics. The key metric used was the *Forecast Error (FE)*, which is the ratio of the absolute forecasting error as a proportion of the time elapsed since the previous MaGC:

$$FE = \frac{(FT - RT)}{(RT - PRT)} \quad (1)$$

where *FT* is the Forecast Time of when the next MaGC will occur, *RT* is the Real Time when the MaGC occurs and *PRT* is the Real Time when the Previous MaGC occurred. *FE=0* means a perfect match between the forecast and the reality. *FE>0* means the real MaGC occurred before the forecast, and *FE<0* means the real MaGC occurred after the forecast. It is usually expressed as a percentage to be comparable among different programs. To illustrate the metric, consider a case where FT was 15 sec since the application started and RT was 14.8 sec. Assuming PRT was 10 sec, *FE* would be 4.17%.

[6]http://dacapobench.org/
[7]http://www.dacapobench.org/benchmarks.html
[8]http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html

Experimental Results. The objective was to assess the accuracy of the forecast algorithm. Even though the results varied among the different GC strategies, it was possible to achieve a *Forecast Error (FE)* below 10% for all the benchmarks. These results are presented in Figure 5.

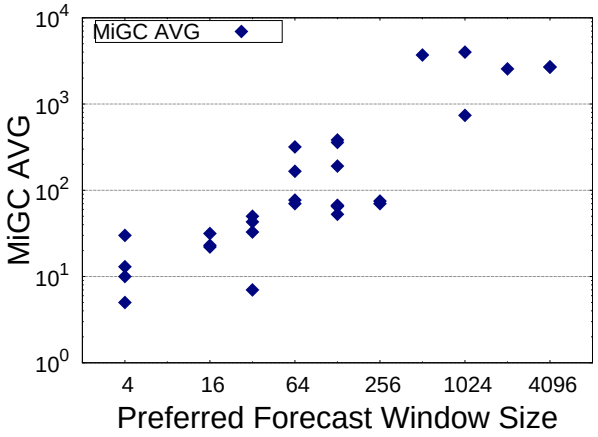

Fig. 6. Preferred FWS vs. MiGC AVG

As no single FWS achieved the lowest FE for all benchmarks, the analysis centered in understanding the factors behind the preferred FWS. As an initial step, the results were sorted by the average number of MiGCs between MaGCs ($MiGC_{AVG}$). This criterion was chosen because it captures the relationship between the allocation needs of an application and the heap size (major factors influencing the GC, as proved by [8] and [13] respectively). The smaller the $MiGC_{AVG}$ is, the more frequent the application exhausts its *Old Generation* memory. If the value is close to zero (i.e. 5 or less), the application is close to an Out-Of-Memory exception. On the contrary, a value far from zero (i.e. 1,000 or more) indicates that the *Old Generation* is infrequently exhausted. The results showed a relationship between the $MiGC_{AVG}$ and the preferred *FWS*: If an application has a high $MiGC_{AVG}$, a large FWS is preferred because a small one does not capture the behaviour of the allocations in the *Old Generation*, which happens infrequently. Similarly, if an application has a low $MiGC_{AVG}$, a small FWS works better. This tendency is visually shown in Figure 6 and experimentally proved in Figure 7.

To further explore the sensitivity of the algorithm to the FWS, the results were analyzed with the coefficient of variation[9] $MiGC_{CV}$ (standard deviation of the $MiGC_{AVG}$ depicted as a percentage of the average) to compare the applications in terms of variability. This analysis showed that the higher the value of $MiGC_{CV}$ (reflecting a more heterogeneous behaviour of the application in terms of memory usage), the more sensitive the algorithm is to changes in FWS. When this occurs, a more precise selection of FWS is required to achieve a low FE. On the contrary, if the $MiGC_{CV}$ is low, a broader range of FWS can be used. Figure 7 exemplifies these two scenarios: h2-Serial GC has a low $MiGC_{AVG}$ (13), so smaller FWS are preferable. As h2 also has a high $MiGC_{CV}$

[9]http://ncalculators.com/statistics/coefficient-of-variance-calculator.htm

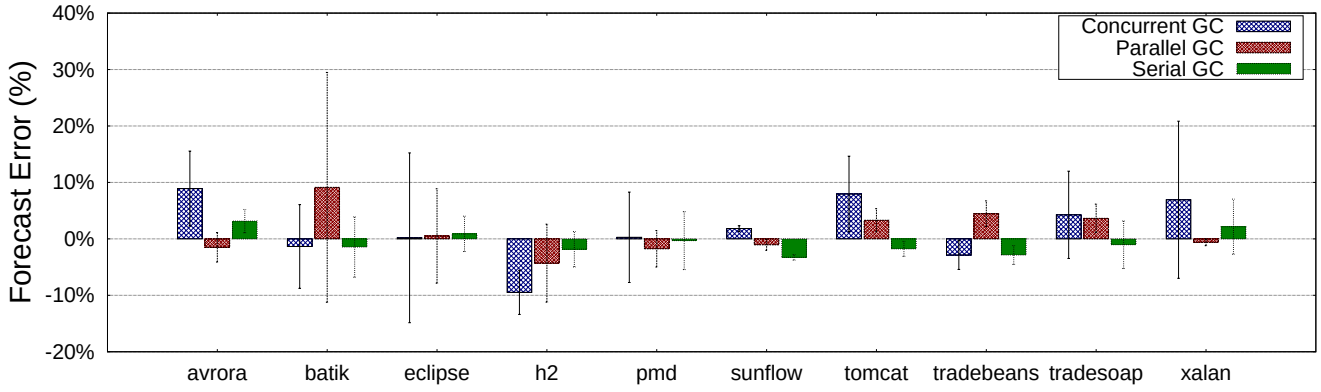

Fig. 5. Forecast Error per DaCapo Benchmark and GC strategy

(70%), it requires a more exact FWS range (between 2 and 16). On the contrary, larger FWS are preferable for tomcat-Serial GC because it has a high $MiGC_{AVG}$ (12673). As tomcat also has a low $MiGC_{CV}$ (7%), a low FE can be achieved using a broader FWS range (between 1024 and 4096).

In conclusion, this experiment proved that the forecast algorithm can achieve a low FE (below 10%) when configured properly. Also two relevant factors to consider in the selection of the FWS ($MiGC_{AVG}$ and $MiGC_{CV}$) were identified.

B. Experiment #2: MaGC-Aware Load Balancing

Environment. It was composed of seven VMs: Five application nodes, one load balancer and one load tester (using Apache JMeter 2.9[10]). All VMs had the characteristics described in the Experiment #1.

Java Benchmarks. From the DaCapo suite, the two programs closest to our use case were selected (tradebeans and tradesoap). Internally they leverage on the DayTrader benchmark[11] which simulates an online stock trading system. This benchmark ran over a Geronimo Application Server[12] 2.1.4 with a 10GB heap, and an in-memory Derby[13] database.

Load Balance Algorithms. Our algorithm was compared against the normal round robin algorithm. To compensate the *Forecast Error (FE)* of the MaGC forecast, the *MaGC Threshold* was set to the FE_{AVG} of the tested programs (5 seconds). Internally, our forecast algorithm used a FWS of 64.

GC. Among the strategies used in the experiment #1, the two which suffer the longest pauses[15] (benefitting more from our load balance algorithm) were used: Serial and Parallel.

Metrics. Throughput (tps) and response time (ms) were collected with JMeter. The CPU (%) and memory (MB) utilisations of the load balancer were collected with nmon[14].

Experimental Results. The objective was to assess the benefits of load balancing based on the MaGC forecast. Two types of runs were performed for each program and GC strategy: One used the normal round robin algorithm and was considered the *Baseline* (BL). The other type used our load balance algorithm (GCLB). Each run involved 150 concurrent users, lasted approximately 30 minutes and produced around 50,000 transactions. Originally we considered to also compare our algorithm against a reactive strategy, where the workload got adapted once a MaGC occurs. However this strategy could not be implemented because it is not possible to know, from a JVM, when a GC is happening (only when it has ended)[15].

The results proved that considering the MaGC forecast in the load balance logic improves significantly the performance

[10]http://jmeter.apache.org/
[11]http://www.dacapobench.org/daytrader.html
[12]https://geronimo.apache.org/
[13]http://db.apache.org/derby/

[14]http://nmon.sourceforge.net/
[15]http://docs.oracle.com/javase/7/docs/api/java/lang/management/-GarbageCollectorMXBean.html

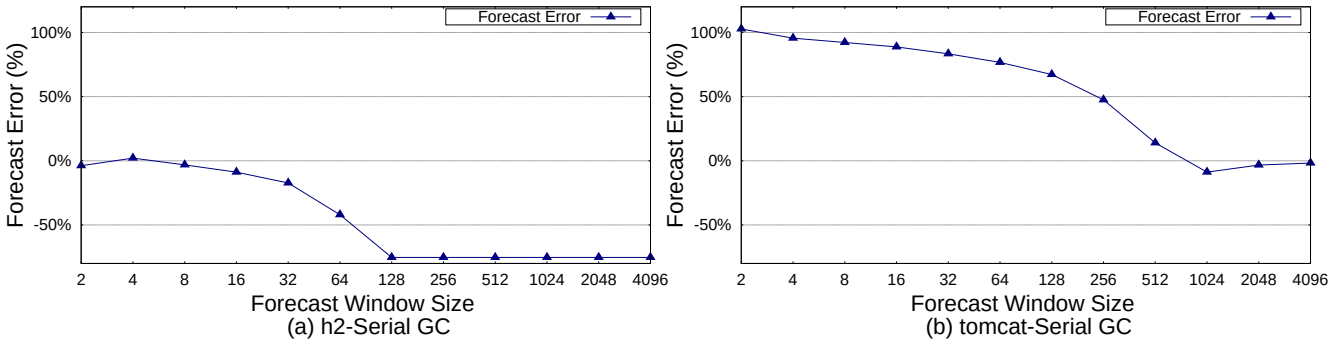

Fig. 7. Forecast Error per FWS for h2 and tomcat benchmarks

TABLE II
THROUGHPUT AND RESPONSE TIME COMPARISON - FULL EXPERIMENT

Bench.	GC	Response Time (ms)						Throughput (tps)					
		RT_{AVG}			RT_{MAX}			T_{AVG}			T_{MIN}		
		BL	GCLB	Diff.(%)	BL	GCLB	Diff.(%)	BL	GCLB	Diff.(%)	BL	GCLB	Diff.(%)
tradebeans	S	4,552.4	112.2	-97.5%	330,813.0	17,596.4	-94.7%	41.4	53.7	29.6%	20.8	38.1	83.5%
tradebeans	P	1,900.1	494.9	-74.0%	305,098.0	33,366.0	-89.1%	46.7	48.5	3.8%	24.8	39.0	57.4%
tradesoap	S	6,757.8	72.0	-98.9%	139,678.0	59,348.9	-57.5%	17.2	25.9	50.6%	11.1	19.2	72.4%
tradesoap	P	845.5	146.0	-82.7%	115,655.0	21,389.7	-81.5%	16.6	17.8	7.4%	5.1	13.1	158.8%

of the system. The average response time (RT_{AVG}) was reduced between 74% and 98.9%, while the maximum response time (RT_{MAX}) was reduced between 57.5% and 94.7%. The throughput experienced a similar improvement: The average throughput (T_{AVG}) increased between 3.8% and 50.6%, while the minimum throughput (T_{MIN}) increased between 57.4% and 158.8%. These results are presented in Table II.

The performance gains were the result of preventing that the MaGCs in the nodes affected the performance of the system. This behaviour is depicted in Figures 8 and 9, which show the results of one of the tested configurations. In Figure 8.a, it can be noticed how the response time of the *Baseline* is affected when a MaGC occurs. On the contrary, Figure 8.b shows that these peaks do not occur when using our algorithm. The throughput (Figure 9) shows a similar behaviour.

To understand better the performance gains of our algorithm over the *Baseline*, the results were analysed under two perspectives. Firstly, the performance was compared during the periods of time when there were no MaGC events (*non-MaGC time*). These results (shown in Table III) proved that our algorithm does not affect the performance of the system during the *non-MaGC time*, as both algorithms performed similarly. Then the performance was compared during the periods of time of the MaGC events (*MaGC time*). These results (shown in Table IV) demonstrated that our algorithm improves the system performance during the *MaGC time*: RT_{AVG} decreased between 87.4% and 99%, while T_{AVG} increased between 42.6% and 97.5%. These improvements were the result of minimising the number of transactions affected by the MaGC. With our algorithm, the only affected transactions were those in the pipeline to be processed by the node which sufferred the MaGC, transactions which led to the triggering of the MaGC.

To understand the costs of our algorithm, we also compared the resource usages in the load balance node. Table V shows these results. The average CPU usage (CPU_{AVG}) increased between 3.5% and 7.2%, and the maximum CPU usage (CPU_{MAX}) between 1.5% and 5.5%. Regarding memory, its average usage (MEM_{AVG}) increased 0.3GB and its maximum usage (MEM_{MAX}) between 0.1 and 0.3GB. These memory increases were caused by the historical information that the forecast algorithm maintained. These increments were considered tolerable because the load balancer was far from exhausting its resources.

In summary, this experiment demonstrated the performance gains of using our proposed algorithm. By avoiding the impact of the MaGCs, the system performance was significantly improved in terms of response time and throughput.

V. CONCLUSIONS AND FUTURE WORK

This paper proposes a new load balancing algorithm to improve the throughput and response time of a distributed system with a small performance overhead. The algorithm utilises JVM data to predict the future occurrences of the MaGC event, which can cause a long pause time on the underlying application. The results have shown that the proposed load balance algorithm can offer a high improvement in response time and throughput (up to 99% and 51% respectively) by using the forecast to decide on how to balance the workload among the system nodes. Furthermore, the proposed algorithm explores and uses a new aspect of the system resource information: The GC. As a result, our work can be combined with other load balancing algorithms to form a more sophisticated solution. This scenario will be explored in our future work, as well as how best to simplify the configuration of our algorithms (e.g., the FWS selection) to improve their applicability.

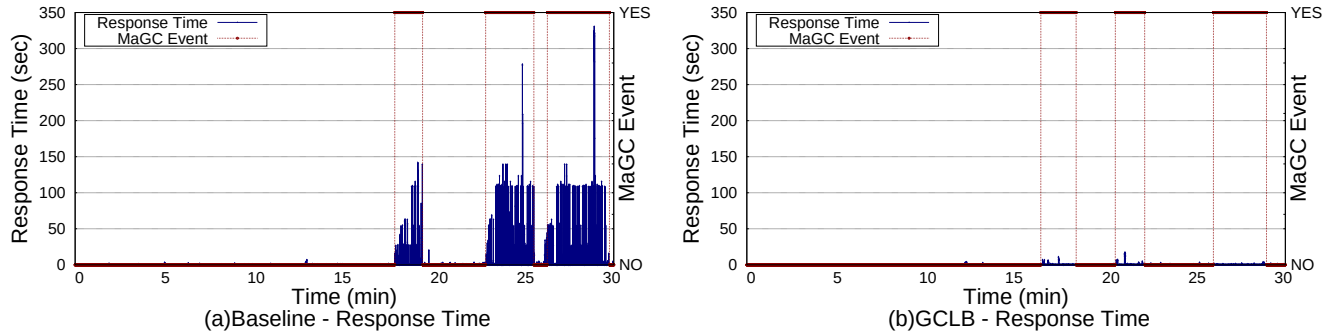

Fig. 8. Performance Comparison - Response Time (tradebeans - Serial GC)

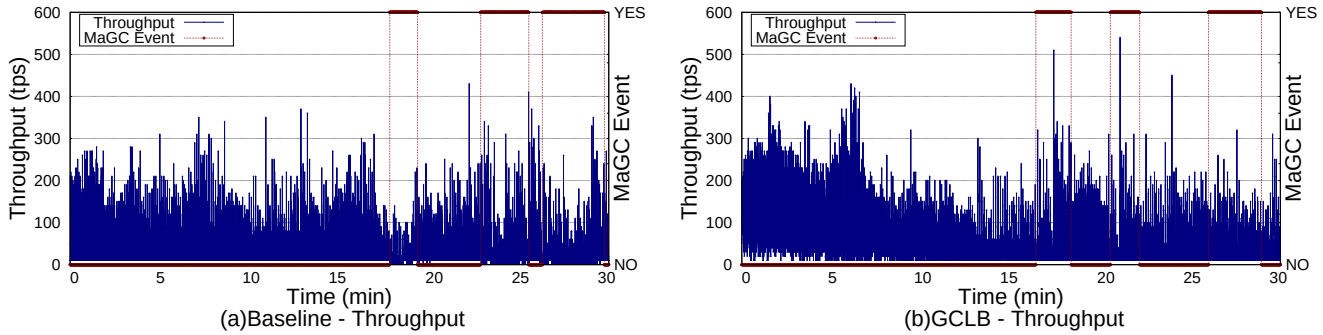

Fig. 9. Performance Comparison - Throughput (tradebeans - Serial GC)

VI. Acknowledgments

Supported, in part, by Science Foundation Ireland grant 10/CE/I1855.

References

[1] S. M. Blackburn and et al. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. *SIGPLAN*, Oct. 2006.
[2] A. B. Carmona, J. Roca-Piera, C. H. Capel, and J. A. Álvarez Bermejo. Adaptive Load Balancing between Static and Dynamic Layers in J2EE Applications. In *NWeSP*, 2011.
[3] H. Cho, C. Na, B. Ravindran, and E. D. Jensen. Scheduling GC in dynamic RT systems with statistical timing assurances. *RTS*, April 2007.
[4] T. Kalibera. Replicating real-time GC for Java. *JTRES*, 2009.
[5] E. Laskowski, M. Tudruj, and R. Olejnik. Dynamic load balancing based on applications global states monitoring. In *ISPDC*, 2013.
[6] Y. Liu, L. Wang, and S. Li. Research on self-adaptive load balancing in EJB clustering system. *ISKE*, 2008.
[7] W. Manning. Scjp sun certified programmer for java 6 exam. *Emereo Pty Ltd, London*, 2009.
[8] F. Mao, E. Z. Zhang, and X. Shen. Influence of program inputs on the selection of garbage collectors. *VEE*, 2009.
[9] G. Phipps. Comparing Observed Bug and Productivity Rates for Java and C++. *Software Practice and Experience*, April 1999.
[10] F. Pizlo, E. Petrank, and B. Steensgaard. A study of concurrent real-time Garbage Collection. *PLDI*, 2008.
[11] L. Rupprecht, A. Reiser, and A. Kemper. Dynamic load balancing in data grids by global load estimation. In *ISPDC*, 2012.
[12] F. Siebert. Limits of parallel GC. *ISMM*, 2008.
[13] J. Singer, R. E. Jones, G. Brown, and M. Luján. The economics of garbage collection. In *ISMM*, 2010.
[14] R. G. Snatzke. Perf. survey. *Codecentric AG*, 2008.
[15] Sun Microsystems. Memory Management in the Java HotSpot Virtual Machine. *April*, 2006.
[16] M. Wegiel and C. Krintz. Dynamic prediction of collection yield for managed runtimes. *SIGPLAN Notices*, Feb. 2009.
[17] P. R. Wilson. Uniprocessor Garbage Collection Techniques. In *International Workshop of Memory Management*, 1992.
[18] F. Xian, W. Srisa-an, H. Jiang, and A. Hall. Garbage Collection : Java Application Servers' Achilles Heel. *SCP*, Feb. 2008.

TABLE III
Throughput and Response Time Comparison - Non-MaGC time

Bench.	GC	Response Time (ms)						Throughput (tps)					
		RT_{AVG}			RT_{MAX}			T_{AVG}			T_{MIN}		
		BL	GCLB	Diff.(%)	BL	GCLB	Diff.(%)	BL	GCLB	Diff.(%)	BL	GCLB	Diff.(%)
tradebeans	S	39.1	31.5	-19.4%	1,953.6	1,948.7	-0.3%	48.3	48.9	1.3%	39.2	40.1	2.3%
tradebeans	P	241.6	336.5	39.3%	1,135.2	1,157.4	2.0%	49.6	49.5	-0.3%	37.1	39.00	5.0%
tradesoap	S	22.3	19.8	-11.2%	287.1	267.9	-6.7%	25.6	26.3	2.5%	16.4	19.2	16.7%
tradesoap	P	123.1	124.4	1.1%	376.8	391.4	3.9%	17.6	17.3	-1.7%	15.5	13.1	-15.2%

TABLE IV
Throughput and Response Time Comparison - MaGC time

Bench.	GC	Response Time (ms)						Throughput (tps)					
		RT_{AVG}			RT_{MAX}			T_{AVG}			T_{MIN}		
		BL	GCLB	Diff.(%)	BL	GCLB	Diff.(%)	BL	GCLB	Diff.(%)	BL	GCLB	Diff.(%)
tradebeans	S	9,065.6	192.8	-97.9%	330,813.0	17,596.4	-94.7%	34.5	50.3	45.5%	20.8	38.1	83.5%
tradebeans	P	10,192.5	1,287.0	-87.4%	305,098.0	33,366.0	-89.1%	29.4	43.5	47.9%	24.8	39.0	57.4%
tradesoap	S	9,163.4	90.6	-99.0%	139,678.0	59,349.0	-57.5%	13.0	25.8	97.5%	11.1	19.2	72.4%
tradesoap	P	3,012.9	210.8	-93.0%	115,655.0	21,389.7	-81.5%	13.6	19.4	42.6%	5.1	13.1	158.7%

TABLE V
Resource Usage Comparison - Load Balancer

Bench.	GC	CPU Usage (%)						Memory Usage (GB)					
		CPU_{AVG}			CPU_{MAX}			MEM_{AVG}			MEM_{MAX}		
		BL	GCLB	Diff.	BL	GCLB	Diff.	BL	GCLB	Diff.	BL	GCLB	Diff.
tradebeans	S	6.0%	9.5%	3.5%	22.0%	23.7%	1.7%	2.50	2.80	0.30	2.60	2.90	0.30
tradebeans	P	7.7%	12.2%	4.5%	24.2%	25.7%	1.5%	2.50	2.80	0.30	2.60	2.90	0.30
tradesoap	S	4.2%	10.6%	6.4%	9.8%	15.3%	5.5%	2.50	2.80	0.30	2.60	2.80	0.20
tradesoap	P	4.2%	11.4%	7.2%	11.8%	16.0%	4.2%	2.50	2.80	0.30	2.70	2.80	0.10